INWG # 39
NIC # 18764

V. Cerf/R. Kahn

To: INWG Mailing List
From: V. Cerf and R. Kahn
Subject: HOST and PROCESS Level Protocols for Internetwork Communication

The enclosed document is an attempt to collect and integrate the ideas uncovered at the June, 1973 INWG meeting in New York, as well as some ideas which have been worked out since that time by various other people, some of whom are not members of INWG but have expressed considerable interest in our problems.*

The text of the document is in DRAFT form and we hope that, by presenting this material in its unpolished form, we will stimulate critical remarks and perhaps even counter-proposals from INWG.

Comments and criticisms should be addressed to Vint Cerf, ERL 407, Stanford University, Stanford, California.

* The June meeting included E. Aupperle, R. Metcalfe, R. Scantlebury, D. Walden, and H. Zimmerman. G. Grossman and G. LeLann made contributions after that meeting.

RECEIVED AT NIC 9-13-73

D R A F T C O P Y

Towards Protocols for Internetwork Communication

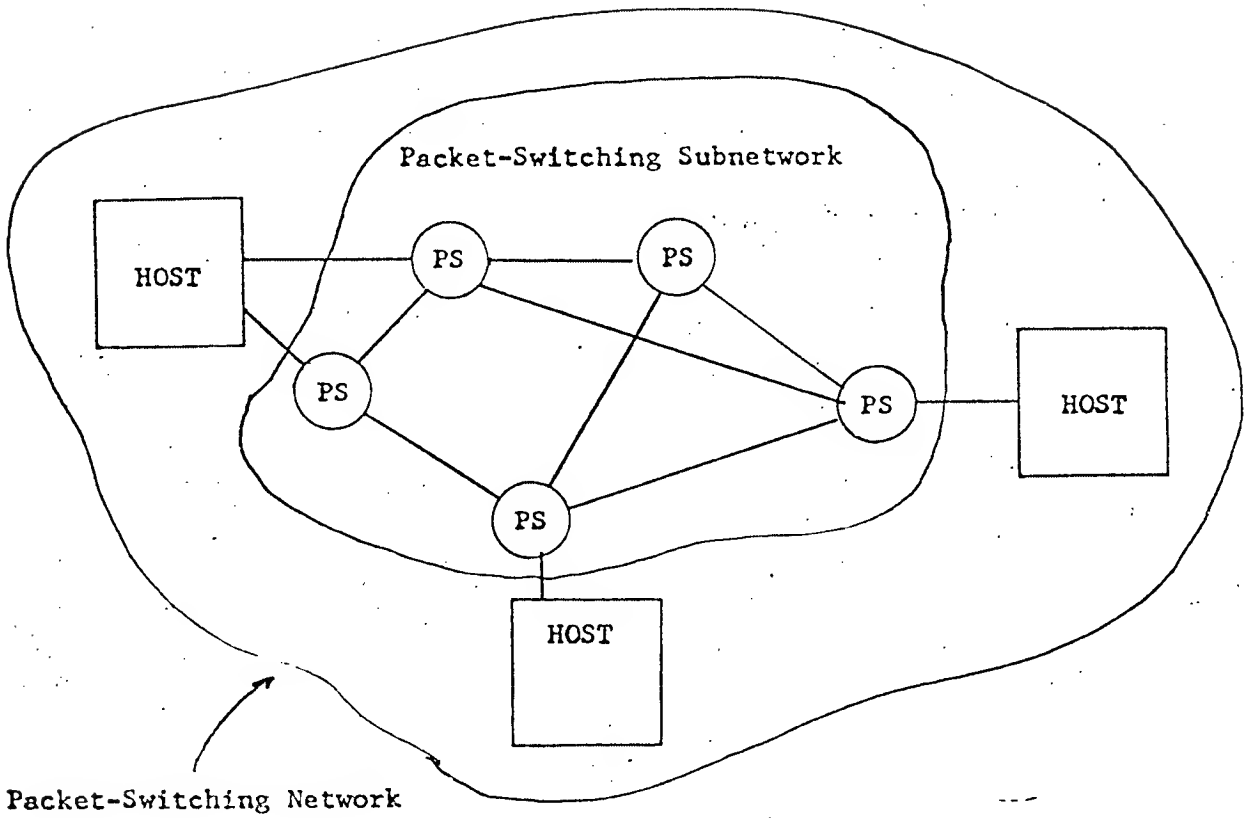
INTRODUCTION

The interconnection of individual packet switching networks into a composite network requires that procedures be developed for their mutual interaction and cooperation. Even though many different and complex problems must be solved in the design of an individual packet switching network, these problems are manifestly compounded when the interconnection is between dissimilar networks. Issues arise which may have no direct counterpart in an individual network and which strongly influence the way in which internetwork communication can take place. In this paper, we discuss some of these issues and formulate a protocol design and philosophy which provides for communication through such a composite network.

We make the assumption that the reader is familiar with packet switching techniques (1, 7, 14, 17), previous work on protocol development (8-12, 16), and their implementation in various individual networks (2-6).

A typical packet switching network is composed of a set of computer resources which are called HOSTs, a set of (one or more) NODEs which serve as packet switches, and a collection of communication media which interconnect the packet switches. Within each HOST, we assume that there exist processes which must communicate with processes in their own or other HOSTs. These processes are generally the ultimate source and destination of data in the network. Any current definition of a process will be adequate for our purposes (13). The ensemble of packet switches and communication media is called the packet switching subnet. Figure 1 illustrates these ideas.

In a typical packet switching subnet, data of a fixed maximum size is accepted from a source HOST, together with a formatted destination address which is used to route the data in a store and forward fashion. The transmit time for this data is typically dependent upon internal network parameters such as data rates, buffering and signalling strategies, routing, propagation delays, etc. In addition, some mechanism is generally present



A typical Packet Switching Network

Figure 1

for error handling anddetermination of status.

To be meaningful, communication between a source and destination process must employ some protocol, which is merely a set of mutually agreed upon conventions. An individual packet switching network, however, does not require knowledge of these conventions for communication to take place. Typically, within an individual network, there exists a protocol for communication between any source and destination process. Processes in two distinct networks would ordinarily use different protocols for this purpose. It is the differences between networks which must be resolved to allow successful communication between them. For example,

1) Each network may have distinct ways of addressing the receiver, thus requiring that a uniform addressing scheme be created which can be understood by each individual network.

2) Each network may accept data of different maximum size, thus requiring networks to deal in units of the smallest maximum size (which may be impractically small), or requiring procedures which allow data crossing a network boundary to be reformatted into smaller pieces.

3) The success or failure of a transmission, and its performance in each network is governed by different time delays in accepting, delivering and transporting the data. This requires careful development of internetwork timing procedures to insure that data can be successfully delivered though the various networks.

4) Within each network, communication my be disrupted due to unrecoverable mutation of the data or missing data. End-to-end restoration procedures are desirable to allow complete recovery from these conditions.

5) Status information, routing, fault detection and isolation are typically different in each network. Thus, to obtain verification

of certain conditions, such as an inaccessible or dead destination, various kinds of co-ordination must be invoked between the communicating networks.

It would be extremely convenient if all the differences between networks could be economically resolved by suitable interfacing at the network boundaries. For many of the differences, this objective can be achieved. However, both economic and technical considerations lead us to prefer that the interface be as simple and reliable as possible and deal primarily with passing data between networks which use different packet switching strategies.

The question now arises as to whether the interface ought to account for differences in HOST or process level protocols by transforming the source conventions into the corresponding destination conventions. We obviously want to allow conversion between packet switching strategies at the interface, to permit interconnection of existing and planned networks. However, the complexity and dissimilarity of the HOST or process level protocols makes it desirable to avoid having to transform between them at the interface, even if this transformation were always possible. Rather, compatible HOST and process level protocols must be developed to achieve effective internetwork resource sharing. The unacceptable alternative is for every HOST or process to implement every protocol (a potentially unbounded number) that may be needed to communicate with other networks. We therefore assume that a common protocol is to be used between HOSTs or processes in different networks and that the interface between networks should take as small a role as possible in this protocol.

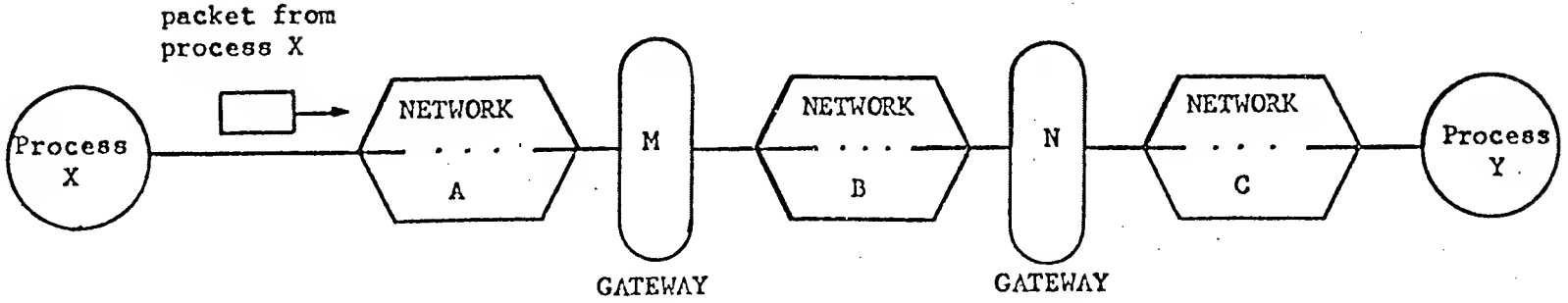
To allow networks under different ownership to interconnect, some accounting may be needed for traffic that passes across the interface. Furthermore, the interconnection must preserve intact the internal operation of each individual network. It is thus apparent that the interface between networks must play a central role in the development of any network interconnection strategy. We give a special name to this interface which performs these functions and call it a GATEWAY.

The GATEWAY Notion

In figure 2, we illustrate three individual networks labelled A, B, and C which are joined by GATEWAYs M and N. GATEWAY M interfaces network A with network B, and GATEWAY N interfaces network B to network C. We assume that an individual network may have more than one GATEWAY (e.g. network B) and there may be more than one GATEWAY path to use in going between a pair of networks. The responsibility for properly routing data resides in the GATEWAY.

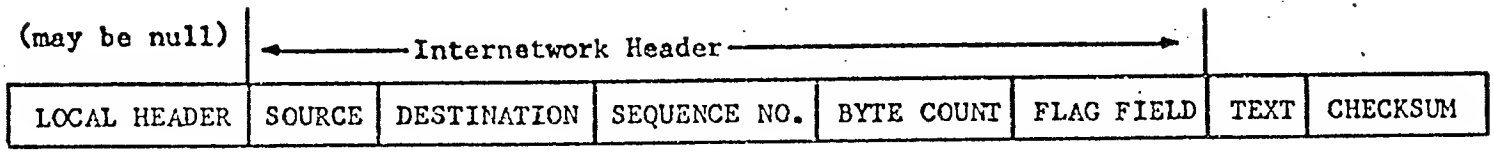
Let us now trace the flow of data through the interconnected networks. We assume a packet of data from process X enters network A destined for process Y in network C. The address of Y is initially specified by process X and the address of GATEWAY M is derived from the address of process Y. We make no attempt to specify whether the choice of GATEWAY is made by process X, its HOST, or one of the packet switches in network A. The packet traverses network A until it reaches GATEWAY M. At the GATEWAY, the packet is reformatted to meet the requirements of network B, account is taken of this unit of flow between A and B, and the GATEWAY delivers the packet to network B. Again the derivation of the next GATEWAY address is accomplished based on the address of the destination, Y. In this case, GATEWAY N is the next one. The packet traverses network B until it finally reaches GATEWAY N where it is formatted to meet the requirements of network C. Account is again taken of this unit of flow between networks B and C. Upon entering network C, the packet is routed to the HOST in which process Y resides and there it is delivered to its ultimate destination.

Since the GATEWAY must understand the address of the source and destination HOSTs, this information must be available in a standard format in every packet which arrives at the GATEWAY. This information is contained in an internetwork header prefixed to the packet. The packet format, including the internetwork header, is illustrated in figure 3. The source and destination entries uniformly and uniquely identify the address of every HOST in the composite network. Addressing is a subject of considerable complexity which is discussed in greater detail in the next section. The next two entries in the header provide



Three Networks Interconnected by Two GATEWAYs

Figure 2



Internetwork Packet Format (fields not shown to scale)

Figure 3

a sequence number and a byte count that may be used to properly sequence the packets upon delivery to the destination and may also enable the GATEWAYs to detect fault conditions affecting the packet. The flag field is used to convey specific control information and is discussed in the section on retransmission and duplicate detection later. The remainder of the packet consists of text for delivery to the destination and a trailing checksum used for end-to-end software verification. The GATEWAY does not modify the text and merely forwards the checksum along without computing or recomputing it.

Each network may need to augment the packet format before it can pass through the individual network. We have indicated a local header in the figure which is prefixed to the beginning of the packet. This local header is introduced merely to illustrate the concept of embedding an internetwork packet in the format of the individual network through which the packet must pass. It will obviously vary in its exact form from network to network, and may even be unnecessary in some cases.

Unless all transmitted packets are legislatively restricted to be small enough to be accepted by every individual network, the GATEWAY may be forced to split a packet into two or more smaller packets. This must be done in such a way that the destination is able to piece together the fragmented packet. We believe it to be undesirable to restrict the size of the internetwork packets to the smallest maximum size available and therefore conclude that the GATEWAYs must be prepared to break up packets into smaller pieces when necessary. It is conceivable that one might desire the GATEWAY to perform the reassembly to simplify the task of the destination HOST (or process) and/or to take advantage of a larger packet size. We take the position that GATEWAYs should not perform this function since GATEWAY reassembly can lead to serious buffering problems, potential deadlocks, the necessity for all fragments of a packet to pass through the same GATEWAY, and increased delay in transmission. Furthermore, it is not sufficient for the GATEWAYs to provide this function since the final

GATEWAY may also have to fragment a packet for transmission. Thus, the destination HOST must be prepared to do this task.

Process Level Communication

We suppose that processes wish to communicate in full duplex with their correspondents using unbounded but finite length messages. A single character might constitute the text of a message from a process to a terminal or vice versa. An entire page of characters might constitute the text of a message from a file to a process. A data stream (e.g. a continuously generated bit string) can be represented as a sequence of finite length messages.

Within a HOST we assume the existence of a Transmission Control Program (TCP) which handles the transmission and acceptance of messages on behalf of the processes it serves. The TCP is in turn served by one or more packet switches connected to the HOST in which the TCP resides. Processes which want to communicate present messages to the TCP for transmission, and TCP's deliver incoming messages to the appropriate destination processes. We allow the TCP to break up messages into segments because the destination may restrict the amount of data which may arrive, because the local network may limit the maximum transmission size, or because the TCP may need to share its resources among many processes concurrently. Furthermore, we constrain the length of a segment to an integral number of 8 bit bytes. This uniformity is most helpful in simplifying the software needed with HOST machines of different natural word lengths. Provision at the process level can be made for padding a message which is not an integral number of bytes and for identifying which of the arriving bytes of text contain information of interest to the receiving process.

Multiplexing and demultiplexing of segments among processes are fundamental tasks of the TCP. On transmission, a TCP must multiplex together segments and produce internetwork packets for delivery to one of its serving packet switches. On reception, a TCP will accept a sequence of packets from its serving packet switch(es). From this sequence of arriving packets, the TCP must be able to reconstruct and deliver messages to the proper destination processes.

We assume that every segment is augmented with additional information which allows transmitting and receiving TCP's to identify destination and source processes respectively. At this point, we must face a major issue. How should the source TCP format segments destined for the same destination TCP? We consider two cases:

Case a) If we take the position that segment boundaries are immaterial and that a byte stream can be formed of segments destined for the same TCP, then we may gain improved transmission efficiency and resource sharing by arbitrarily parceling the stream into packets, permitting many segments to share a single internetwork packet header. However, this position results in the need to reconstruct exactly and in order, the stream of text bytes produced by the source TCP. At the destination, this stream must be parsed into segments and these must in turn be used to reconstruct messages for delivery to the appropriate processes.

There are fundamental problems associated with this strategy due to the possible arrival of packets out of order at the destination. The most critical problem appears to be the amount of interference that processes sharing the same TCP-TCP byte stream may cause among themselves. This is expecially so at the receiving end. First, the TCP may be put to some trouble to parse the back stream into segments, and then distribute them to buffers where messages are reassembled. If it is not readily apparent that all of a segment has arrived (remember, it may come as several packets), the receiving TCP may have to suspend parsing temporarily until more packets have arrived. Second, if a packet is missing, it may not be clear whether succeeding segments, even if they are identifiable, can be passed on to the receiving process, unless the TCP has knowledge of some process level sequencing scheme. Such knowledge would permit the TCP to decide whether a succeeding segment could be delivered to its waiting process. Finding the beginning of a segment when there are gaps in the byte stream may also be hard.

Case b) Alternatively, we might take the position that the destination TCP should be able to determine upon its arrival and without additional information for which process or processes a received packet is intended, and if so, whether it should be delivered then.

If the TCP is to determine for which process an arriving packet is intended, every packet must contain a process header (distinct from the internetwork header) which completely identifies the destination process. For simplicity, we assume that each packet contains text from a single process which is destined for a single process. Thus each packet need contain only one process header. To decide whether the arriving data is deliverable to the destination process, the TCP must be able to determine whether the data is in the proper sequence (we can make provision for the destination process to instruct its TCP to ignore sequencing, but this is considered a special case). With the assumption that each arriving packet contains a process header, the necessary sequencing and demultiplexing information is immediately available to the destination TCP.

Both cases (a) and (b) provide for the demultiplexing and delivery of segments to destination processes, but only case (b) does so without the introduction of potential interprocess interference. For this reason, we select the method of case (b) as a part of the internetwork transmission protocol.

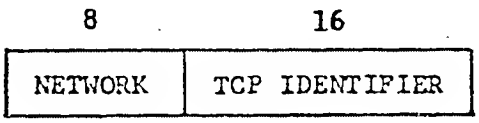
Address Formats

The selection of address formats is a problem between networks because the local network addresses of TCP's may vary substantially in format and size. A uniform internetwork TCP address space, understood by each GATEWAY and TCP, is essential to routing and delivery of internetwork packets.

Similar troubles are encountered when we deal with process and port addressing. We introduce the notion of ports in order to permit a process to distinguish between multiple message streams. The port is simply a designator of one such message stream associated with a process. The means for identifying a process and port are generally different in different operating systems and therefore, to obtain uniform addressing, a standard process/port address format is also required. A process/port address designates a full duplex message stream.

•TCP Addressing

TCP addressing is intimately bound up in routing issues, since a HOST or GATEWAY must choose a suitable destination HOST or GATEWAY for an outgoing internetwork packet. Let us postulate the following address format for the TCP address (figure 4 below):



TCP Address

Figure 4

The choice for network identification (8 bits) allows up to 256 distinct networks. This size seems sufficient for the forseeable future. Similarly, the TCP identifier field permits up to 65,536 distinct TCP's to be addressed which seems more than sufficient for any given network.

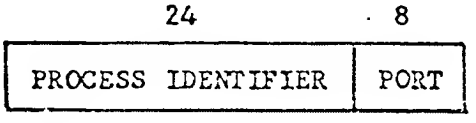
As each packet passes through a GATEWAY, the GATEWAY observes the destination network ID to determine how to route the packet. If the destination network is connected to the GATEWAY, the lower 16 bits of the TCP address are used to produce a local TCP address in the destination network. If the destination network is not connected to the GATEWAY, the upper 8 bits are used to select a subsequent GATEWAY. We make no effort to specify how each individual network shall associate the internetwork TCP identifier with its local TCP address. We also do not rule out the possibility that the local network understands the internetwork addressing scheme and thus alleviates the GATEWAY of the routing responsibility.

•Process/Port Addressing

A receiving TCP is faced with the task of demultiplexing the stream of internetwork packets it receives and reconstructing the original messages for each destination process. Each operating system has its own internal means of identifying processes and ports, and we postulate that 32 bits should be sufficient to serve as internetwork process/port identifiers. A sending process need not know how the destination process/port identification will be used. The destination TCP will be able to parse this number appropriately to find the proper buffer into

which it will place arriving packets. The 32 bits have been divided up in figure 5 into a process identifier and a port number each with 24 bits and 8 bits respectively. We permit a large port number field to support processes which want to distinguish between many different message streams concurrently. In reality, we do not care how the 32 bits are sliced up by the TCP's involved.

To simplify this discussion, we assume that the full process/port identifier accompanies each packet. The use of short names for process/port identifiers is often desirable to reduce transmission overhead and possibly reduce packet processing time at the destination TCP. Assigning double names to each process/port, however, requires an initial negotiation between source and destination to agree on a suitable short name assignment, and the subsequent maintenance of conversion tables at both the source and the destination as well.



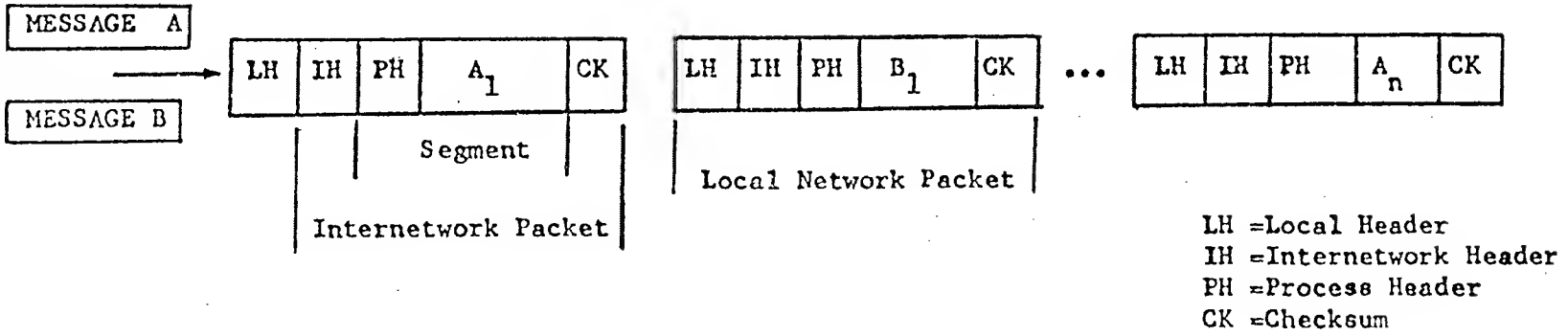
Process/Port Identifier Format

Figure 5

Segment and Packet Formats

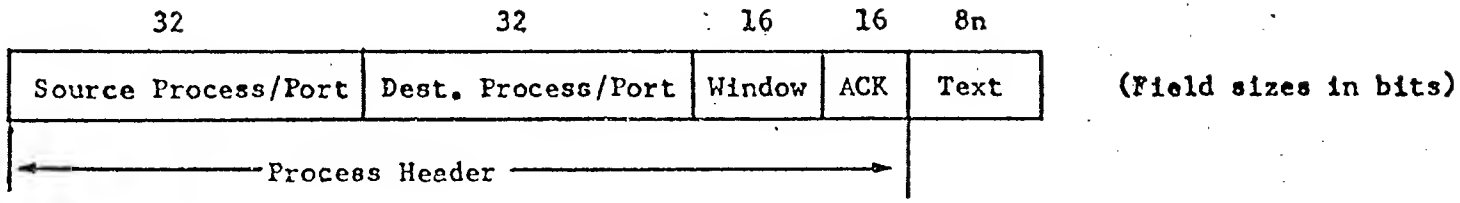
As shown in figure 6, messages are broken by the TCP into segments whose format is shown in more detail in figure 7. The field lengths illustrated are merely suggestive. The first two fields (Source Process/Port and Destination Process/Port) have already been discussed in the section on addressing above. The uses of the third and fourth fields (Window and ACKnowledgment) will be discussed later in the section on retransmission and duplicate detection.

We recall, from figure 3, that an internetwork header contains both a sequence number and a byte count, as well as a flag field and a checksum. The uses of these fields are explained below.



Creation of Segments and Packets from Messages

Figure 6



Segment Format (Process Header and Text)

Figure 7

Reassembly and Sequencing

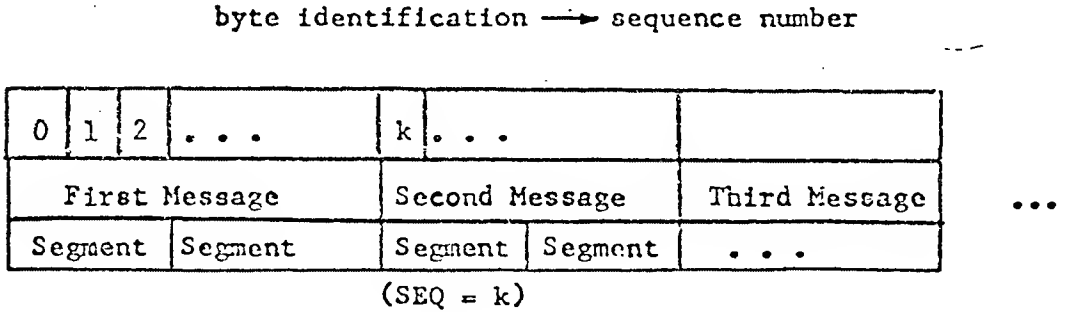
The reconstruction of a message at the receiving TCP clearly requires that each internetwork packet carry a sequence number which is unique to its particular destination Process/Port message stream. The sequence numbers must be monotonic increasing (or decreasing) since they are used to reorder and reassemble arriving packets into a message. If the space of sequence numbers were infinite, we could simply assign the next one to each new packet. Clearly, this space cannot be infinite, and we will consider what problems a finite sequence number space will cause when we discuss retransmission and duplicate detection in the next section. We propose the following scheme for performing the sequencing of packets and hence the reconstruction of messages by the destination TCP.

A pair of Process/Ports will exchange one or more messages over a period of time. We could view the sequence of messages produced by one port as if it were embedded in an infinitely long stream of bytes. Each byte of the message has a unique sequence number which we take to be its byte location relative to the beginning of the stream. When a segment is extracted from the message by the source TCP and formatted for internetwork transmission, the relative location of the first byte of segment text is used as the sequence number for the packet. The byte count field in the internetwork header accounts for all the text in the segment (but does not include the checksum bytes or the bytes in either internetwork or process header). We emphasize that the sequence number associated with a given packet is unique only to the pair of process/ports which are communicating (see figure 8). Arriving packets are examined to determine for which process/port they are intended. The sequence numbers on

each arriving packet are then used to determine the relative location of the packet text in the messages under reconstruction. We note that this allows the exact position of the data in the reconstructed message to be determined even when pieces are still missing.

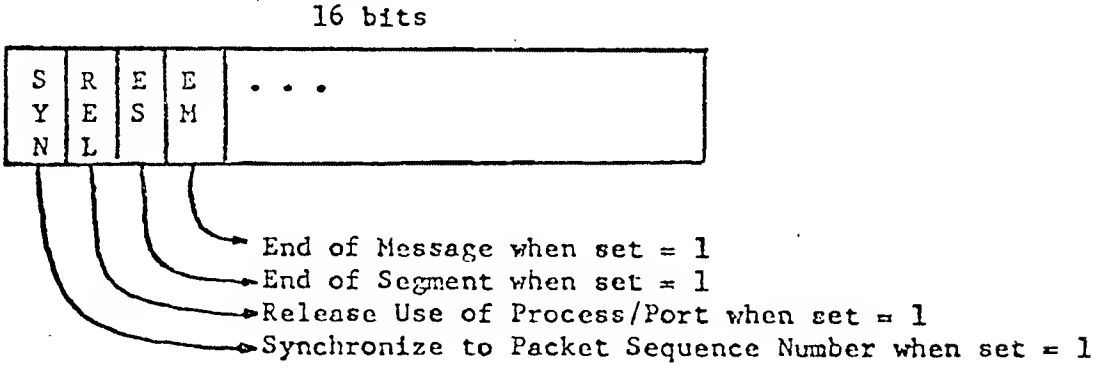
Every segment produced by a source TCP is packaged in a single internetwork packet and a checksum is computed over the text and process header associated with the segment.

The splitting of messages into segments by the TCP and the potential splitting of segments into smaller pieces by GATEWAYs creates the necessity for indicating to the destination TCP when the end of a segment has arrived (ES) and when the end of a message has arrived (EM). The flag field of the internetwork header is used for this purpose (see figure 9).



Assigment of Sequence Numbers

Figure 8



Internetwork Header Flag Field

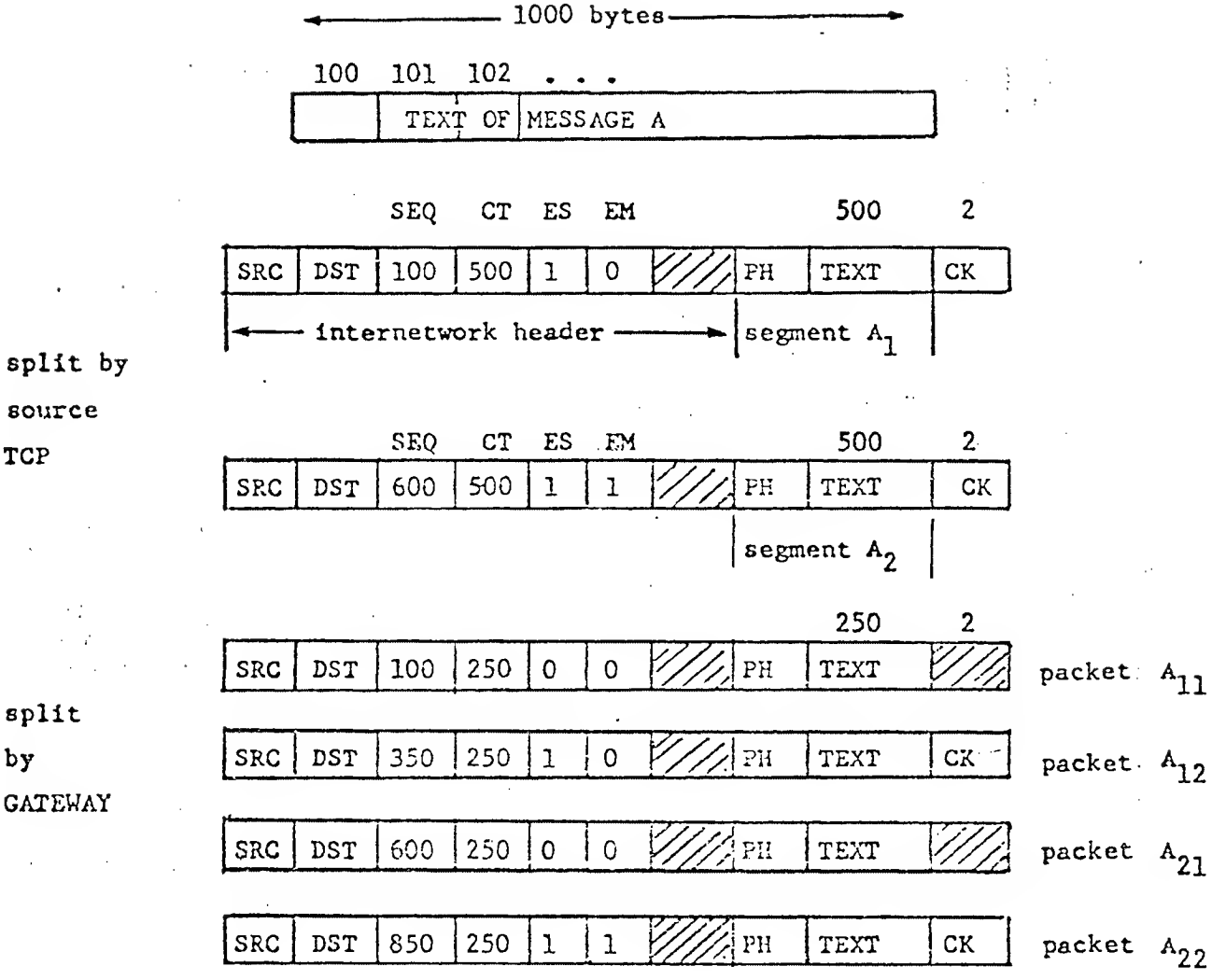
Figure 9

The end of segment flag (ES) is set by the source TCP each time it prepares a segment for transmission. If it should happen that the message is completely contained in the segment, then the end of segment flag (EM) would also be set. The end of message flag is also set on the last segment of a message, if the message could not be contained in one segment. These two flags are used by the destination TCP respectively to discover the presence of a checksum for a given segment and to discover that a complete message has arrived.

The ES and EM flags in the internetwork header are known to the GATEWAY and are of special importance when packets must be split apart for propagation through the next local network. We illustrate their use with an example in figure 10.

The original message, A, in figure 10, is shown split into two segments, A_1 and A_2, and formatted by the TCP into a pair of internetwork packets. Packets A_1 and A_2 have their end of segment bits set and A_2 has its end of message bit set as well. When packet A_1 passes through the GATEWAY, it is split into two pieces: packet A_{11} for which neither end of message nor end of segment bits are set, and packet A_{12} whose end of segment bit is set. Similarly, packet A_2 is split such that the first piece, packet A_{21}, has neither bit set, but packet A_{22} has both bits set. The sequence number field (SEQ) and the byte count field (CT) of each packet is modified by the GATEWAY to properly identify the text bytes of each packet.

The destination TCP, upon reassembling segment A_1, will detect the ES flag and will verify the checksum it knows is contained in packet A_{12}. Upon receipt of packet A_{22}, assuming all other packets have arrived, the destination TCP detects that it has reassembled a complete message and can now advise the destination process of its receipt.



Message splitting and packet splitting

Figure 10

Retransmission and Duplicate Detection

No transmission can be 100% reliable. We propose a time-out and positive acknowledgment mechanism which will allow TCP's to recover from packet losses in the network. A TCP transmits packets and waits for replies (acknowledgments) which are carried in the reverse packet stream. If no acknowledgment for a particular packet is received, the TCP will retransmit.

Any retransmission policy requires some means by which the receiver can detect duplicate arrivals. Even if an infinite number of distinct packet sequence numbers were available, the receiver would still have the problem of knowing how long to remember previously received packets in order to detect duplicates. Matters are complicated by the fact that only a finite number of distinct sequence numbers are in fact available, and if they are re-used, the receiver must be able to distinguish between new transmissions and re-transmissions.

A window strategy, similar to that used by the French CYCLADES system (voie virtuelle transmission mode, 8), is proposed here.

Suppose that the sequence number field in the internetwork header permits sequence numbers to range from 0 to n-1. We assume that the sender will not transmit more than w bytes without receiving an acknowledgment. The w bytes serve as the window (see figure 11). Clearly, w must be less than n. The rules for sender and receiver are as follows:

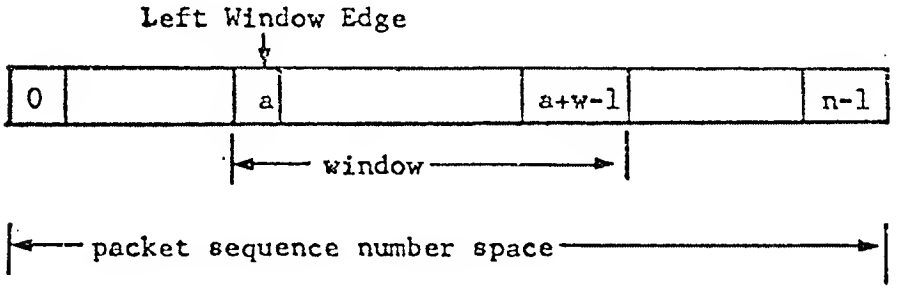
Sender

Let L be the sequence number associated with the left window edge.

1. Transmits bytes from segments whose text lies between L and up to L + w - 1.
2. On timeout (duration unspecified), sender retransmits unacknowledged bytes.
3. On receipt of acknowledgment consisting of the receiver's current left window edge, advances sender left window edge over the acknowledged bytes (advancing the right window edge implitily).

Receiver

1. Arriving packets whose sequence numbers coincide with the receiver's current left window edge are acknowledged by sending to the source the next sequence number expected. This effectively acknowledges bytes in between. The left window edge is advanced to the next sequence number expected.
2. Packets arriving with a sequence number to the left of the window edge (or, in fact, outside of the window) are discarded, and the current left window edge is returned as acknowledgment.
3. Packets whose sequence numbers lie within the receiver's window but do not coincide with the receiver's left window edge are optionally kept or discarded, but are not acknowledged. This is the case when packets arrive out of order.



The Window Concept

Figure 11

We make some observations on this strategy. First, all computations with sequence numbers and window edges must be made modulo n (e.g. byte 0 follows byte n-1). Second, w must be less than n/2, otherwise a retransmission may appear to the receiver to be a new transmission in the case that the receiver has accepted a window's worth of incoming packets, but

all acknowledgments have been lost. Third, the receiver can either save or discard arriving packets whose sequence numbers do not coincide with the receiver's left window. Thus, in the simplest implementation, the receiver need not buffer more than one packet per message stream if space is critical. Fourth, multiple packets can be acknowledged simultaneously. Fifth, the receiver is able to deliver messages to processes in their proper order as a natural result of the reassembly mechanism. Sixth, when duplicates are detected, the acknowledgment method used naturally works to resynchronize sender and receiver. Furthermore, if the receiver accepts packets whose sequence numbers lie within the current window but which are not coincident with the left window edge, an acknowledgment consisting of the current left window edge would act as a stimulus to cause retransmission of the unacknowledged bytes. Finally, we mention an overlap problem which results from retransmission, packet splitting, and alternate routing of packets through different GATEWAYs.

A 600 byte packet might pass through one GATEWAY and be broken into two 300 byte packets. On retransmission, the same packet might be broken into three 200 byte packets going through a different GATEWAY. Since each byte has a sequence number, there is no confusion at the receiving TCP. We leave for later the issue of initially synchronizing the sender and receiver left window edges and the window size.

Flow Control

Every segment which arrives at the destination TCP is acknowledged by returning the sequence number of the next segment which must be passed to the process (it may not yet have arrived). If the segment which has just arrived is the desired next one to be delivered, the acknowledgement will be for the next segment following.

Earlier we described the use of a sequence number space and window to aid in duplicate detection. Acknowledgments are carried in the process header (see figure 7) and along with them there is provision for a "suggested window" which the receiver can use to control the

flow of data from the sender. This is intended to be the main component of the process flow control mechanism. The receiver is free to vary the window size according to any algorithm it desires so long as the window size never exceeds half the sequence number space.

This flow control mechanism is exceedingly powerful and flexible, and does not suffer from synchronization troubles encountered by incremental buffer allocation schemes (9, 10). However, it relies heavily on an effective retransmission strategy. The receiver can reduce the window even while packets are en route from the sender whose window is presently larger. The net effect of this reduction will be that the receiver may discard incoming packets (they may be outside the window) and reiterate the current window size along with a current left window edge as acknowledgment. By the same token, the sender can, upon occasion, choose to send more than a window's worth of data on the possibility that the receiver will expand the window to accept it (of course, the sender must not send more than half the sequence number space at any time). Normally, we would expect the sender to abide by the window limitation. Expansion of the window by the receiver merely allows more data to be accepted. For the receiving HOST with a small amount of buffer space, a strategy of discarding all packets whose sequence numbers do not coincide with the current left edge of the window is probably necessary, but it will incur the expense of extra delay and overhead for retransmission.

TCP Input/Output Handling

The TCP has a component which handles I/O to and from the network. When a packet has arrived, it validates the addresses and places the packet on a queue. A pool of buffers can be set up to handle arrivals, and if all available buffers are used up, succeeding arrivals can be discarded since unacknowledged packets will be retransmitted.

On output, a smaller amount of buffering is needed, since process buffers can hold the data to be transmitted. Perhaps double buffering will be adequate. We make no attempt to specify how the buffering should be done, except to require that it be able to service the network with as little overhead as possible. Packet sized buffers, one or more ring

buffers, or any other combination are possible candidates.

When a packet arrives at the destination TCP, it is placed on a queue which the TCP services frequently. For example, the TCP could be interrupted when a queue placement occurs. The TCP then attempts to place the packet text into the proper place in the appropriate process receive buffer. If the packet terminates a segment, then it can be checksummed and acknowledged. Placement may fail for several reasons:

1. The destination process may not be prepared to receive from the stated source, or the destination port ID may not exist.
2. There may be insufficient buffer space for the text.
3. The beginning sequence number of the text may not coincide with the next sequence number to be delivered to the process (e.g. the packet has arrived out of order).

In the first case, the TCP should simply discard the packet (thus far, no provision has been made for error acknowledgments). In the second and third cases, the packet sequence number can be inspected to determine whether the packet text lies within the legitimate window for reception. If it does, the TCP may optionally keep the packet queued for later processing. If not, the TCP can discard the packet, but acknowledge with the current left window edge.

It may happen that the process receive buffer is not present in the active memory of the HOST, but is stored on secondary storage. If this is the case, the TCP can prompt the scheduler to bring in the appropriate buffer and the packet can be queued for later processing.

If there are no more input buffers available to the TCP for temporary queueing of incoming packets, and if the TCP cannot quickly use the arriving data (e.g. a TCP to TCP messsage), then the packet is discarded. Assuming a sensibly functioning system, no other processes than the one for which the packet was intended should be affected by this discarding. If the delayed processing queue grows excessively long, any packets in it can be safely discarded since none of them have yet been acknowledged. Congestion at the TCP level is flexibly handled owing to the robust retransmission and duplicate detection strategy.

TCP/PROCESS Communication

In order to send a message, a process sets up its text in a buffer region in its own address space, inserts the requisite control information (described below) in a Transmit Control Block (TCB) and passes control to the TCP.. The exact form of a TCB is not specified here, but it might take the form of a passed pointer, a pseudo-interrupt, or various other forms. To receive a message in its address space, a process sets up a receive buffer, inserts the requisite control information in a Receive Control Block (RCB) and again passes control to the TCP.

In some simple systems, the buffer space may in fact be provided by the TCP. For simplicity we assume that a ring buffer is used by each process, but other structures (e.g. buffer chaining) are not ruled out.

A possible format for the TCB is shown in figure 13. The TCB contains information necessary to allow the TCP to extract and send the process data. Some of the information might be implicitly known, but we are not concerned with that level of detail. The various fields in the TCB are described below.

1. Source Address
   This is the full Net/HOST/TCP/Process/Port address of the transmitter.

2. Destination Address
   This is the full Net/HOST/TCP/Process/Port address of the receiver.

3. Next Packet Sequence Number
   This is the sequence number to be used for the next packet the TCP will transmit from this port.

4. Current Buffer Size
   This is the present size of the process transmit buffer.

5. Next Write Position
   This is the address of the next position in the buffer at which the process can place new data for transmission.

6. Next Read Position
   This is the address at which the TCP should begin reading to build the next segment for output.

7. End Read Position
   This is the address at which the TCP should halt transmission. Initially, (6) and (7) bound the message which the process wishes to transmit.

8. Number of Retransmissions/ Maximum Retransmissions
   These fields enable the TCP to keep track of the number of times it has retransmitted the data and could be omitted if the TCP is not to give up.

9. Timeout/Flags
   The timeout field specifies the delay after which unacknowledged data should be retransmitted. The Flag field is used for semaphores and other TCP/Process synchronization, status reporting, etc.

10. Current Acknowledgement/Window
    The current acknowledgment field identifies the first byte of data still unacknowledged by the destination TCP.

1	Source Address	
2	Destination Address	
3	Next Packet Seq.	
4	Current Buffer Size	
5	Next Write Position	
6	Next Read Position	
7	End Read Position	
8	# Retrans.	Max Retrans.
9	Timeout	Flags
10	Curr. Ack	Window

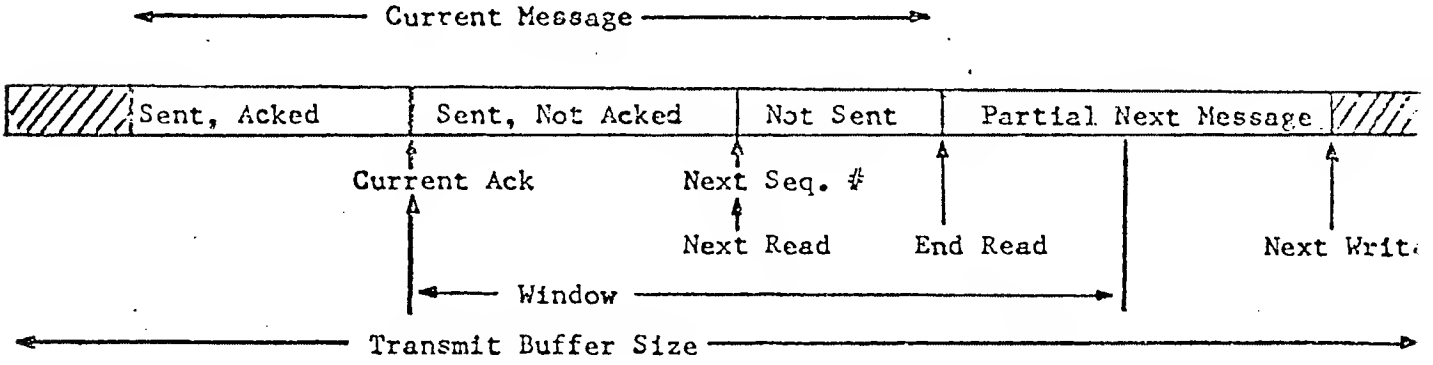
Conceptual TCB Format

Figure 13

The read and write positions move circularly around the transmit buffer, with the write position always to the left (module the buffer size) of the read position.

The next packet sequence number should be constrained to be less than or equal to the sum of the current acknowledgment and the window fields. In any event, the next sequence number should not exceed the sum of the current acknowledgment and half of the maximum possible sequence number (to avoid confusing the receiver's duplicate detection algorithm). A possible buffer layout is shown in figure 14.

The receive control block (RCB) is substantially the same, except that the end read field is replaced by a partial segment checksum register which permits the receiving TCP to compute and remember partial checksums in the event that a segment arrives in several packets. When the final packet of the segment arrives, the TCP can verify the checksum and if successful, acknowledge the segment.



Transmit Buffer Layout

Figure 14

Partners and Associations

Much of the thinking about process-to-process communication in packet switched networks has been influenced by the ubiquitous telephone system. The HOST-HOST protocol for the ARPANET deals explicitly with the opening and closing of simplex connections between processes (9, 10). Evidence has been presented that message-based, "connection-free" protocols can be constructed (12)and this leads us to carefully examine the notion of a connection.

The term connection has a wide variety of meanings. It can refer to a physical or logical path between two entities, it can refer to the flow over the path, it can inferentially refer to an action associated with the setting up of a path, or it can refer to an association between two or more entities, with or without regard to any path between them. In this paper, were we to adopt the term connection, we would use it in the sense of an association between two or more entities without regard to a path. To be more precise about our intent, we shall define the relationship between two or more process/ports that are in communication, or are prepared to communicate to be an association. Process/ports that are associated with each other are called partners.

It is clear that for any communication to take place between two processes, one must be able to address the other. The two important cases here are that the destination process may have a global and unchanging address or that it may be globally unique but dynamically reassigned. While in either case, the sender may have to learn the destination address, given the destination name, only in the second instance is there a requirement for learning the address from the destination (or its representative) each time an association is desired. Only after the source has learned how to address the destination can an association be said to have occurred. But this is not yet sufficient. If ordering of delivered messages is also desired, both TCP's must maintain sufficient information to allow proper sequencing. When this information is also present at both ends, then an association is said to have occurred.

Note that we have not said anything about a path, nor anything which implies that either end be aware of the condition of the other. Only when both partners are prepared to communicate with each other has an association occurred, and it is possible that neither partner may be able to verify that an association exists until some data flows between them.

Connection-free protocols with associations

In the ARPANET, the Interface Message Processors (IMPs) do not have to open and close connections from source to destination. The reason for this is that connections are, in effect, always open, since the address of every source and destination is never* re-assigned. When the name and the place are static and unchanging, it is only necessary to label a packet with source and destination to transmit it through the network. In our parlance, every source and destination forms an association.

In the case of processes, however, we find that process/port addresses are continually being used and re-used. Some ever-present processes could be assigned fixed addresses which do not change (e.g. the Logger Process). If we supposed, however, that every TCP had an infinite supply of process addresses so that no old address would ever be re-used, then any dynamically created process would be assigned the next unused address. In such an environment, there could never be any confusion by source and destination TCP as to the intended recipient or implied source of each message, and all processes would be partners.

Unfortunately, TCP's (or more properly, operating systems) tend not to have an infinite supply of internal process addresses. These internal addresses are re-assigned after the demise of each process. Walden (12) suggests that a set of unique, uniform, external process addresses could be supplied by a central registry. A newly

---

* Unless the IMP is physically moved to another site

created process could apply to the central registry for an address which the central registry would guarantee to be unused by any HOST system in the network. Each TCP could maintain tables matching external names with internal ones, and use the external ones for communication with other processes. This idea violates the premise that interprocess communication should not require centralized control. One would have to extend the central registry service to include all HOSTs in all the interconnected networks to apply this idea to our situation, and we therefore do not attempt to adopt it.

Let us consider the situation from the standpoint of the TCP. In order to send or receive data for a given Process/Port, the TCP needs to set up a TCB and RCB, and initialize the window size and left window edge for both. On the receive side, this task might even be delayed until the first packet destined for a given Process/Port arrives. By convention, the first packet should be marked so that the receiver will synchronize to the received sequence number.

On the send side, the first request to transmit could cause a TCB to be set up with some initial sequence number (say zero) and an assumed window size. The receiving TCP can reject the packet if it wishes and notify the sending TCP of the correct window size via the acknowledgment mechanism, but only if either

a) we insist that the first packet be a complete segment
b) an acknowledgment can be sent for the first packet (even if not a segment) as long as the acknowledgment specifies the next sequence number such that the source also understands that no bytes have been accepted.

It is apparent, therefore, that the synchronizing of window size and left window edge can be accomplished without what would ordinarily be called a connection set-up.

The first packet referencing a newly created RCB sent from one partner to another can be marked with a bit which requests that the receiver synchronize his left window edge with the sequence number of the arriving packet (see SYN bit in figure 9). The TCP can examine the source and destination Process/Port addresses in the packet and in the RCB to decide whether to accept or ignore the request.

Provision should be made for a destination process to specify that it is willing to "listen" to a specific partner or "ANY" partner. This last idea permits processes such as the Logger Process to accept data arriving from unspecified sources.

The initial packet may contain data which can be stored or discarded by the destination, depending on the availability of destination buffer space at the time. In the other direction, acknowledgment is returned for receipt of data which also specifies the receiver's window size.

If the receiving TCP should want to reject the synchronization request, it merely transmits an acknowledgment carrying a RELease bit (see figure 9) indicating that the destination Process/Port address is unknown or inaccessible. This rejection is quite different from a negative data acknowledgment. We do not have explicit negative acknowledgments.

Because messages may be broken up into many packets for transmission or during transmission, it will be necessary to ignore the REL flag except in the case that the EM (end of message) flag is also set. This could be accomplished either by the TCP or by the GATEWAY which could reset the flag on all but the packat containing the set EM flag (see figure 10).

At the end of an association, the TCP sends a packet with ES, EM, and REL flags set. The packet sequence number scheme will alert the receiving TCP if there are still outstanding packets in transit which have not yet arrived, so a premature disassociation cannot occur.

To assure that both TCPs are awayre that the association has ended, we insist that the receiving TCP respond to the REL by sending an REL acknowledgment of its own.

Suppose now that a process sends a single message to a partner including an REL along with the data. Assuming an RCB has been prepared for the receiving TCP to accept the data, the TCP will accumulate the incoming packets until the one marked ES, EM, REL

arrives, at which point an REL is returned to the sender. The association is thereby terminated and the appropriate TCB and RCB are destroyed. If the first packet of a message contains a SYNc request bit and the last packet contains ES, EM, and REL bits, then data will flow "one message at a time." This node is very similar to the scheme described by Walden (12), since each succeeding message can only be accepted atthe receiver after a new Listen (like Walden's "IN") command is issued by the receiving process to its serving TCP. Alternatively, both processes can send one message, causing the respective TCP's to allocate RCB/TCB pairs at both ends which rendezvous with the exchanged data and then disappear. If the overhead of creating and destroying RCB's and TCB's is small, such a protocol might be adequate for most low bandwidth uses. This idea might also form the basis for a relatively secure transmission system. If the communicating processes agree to change their external addresses in some way known only to each other (ie. pseudo-random), then each message will appear to the outside world as if it is part of a different association message stream. Even if the data is intercepted by a third party, it will have no way of knowing that the data should in fact be considered part of a sequence of messages.

We have described the way in which processes develop associations with each, thereby becoming partners for possible exchange of data. These associations need not involve the transmission of data prior to their formation and indeed two partners need not be able to determine that they are partners until they attempt to communicate.

Conclusions

We have uncovered some fundamental issues related to the interconnection of packet switching networks. In particular, we have described a simple but very powerful and flexible protocol which provides for variation in individual network packet sizes, transmission failures, sequencing, flow control, and the creation and destruction of process-to-process associations. We have considered some of the implementation issues that arise and

found that the proposed protocol is implementable by HOSTs of widely varying capacity. The tradeoffs between transmission overhead (large headers) and TCP table manipulation/storage requirements have been considered in some depth.

The next important step is to produce a detailed specification of the protocol so that some initial experiments with it can be performed. These experiments are needed to determine some of the operational parameters (e.g. how often and how far out of order do packets actually arrive? What sort of delay is there between segment acknowledgements? What should the retransmission timeouts be?).

13. Lampson, B., "A Scheduling Philosophy for Multiprocessing Systems," Comm. of the ACM, 11,5, May 1968, pp 347-360.

14. Heart, F. E. and R. Kahn, S. Ornstein, W. Crowther, D. Walden, "The Interface Message Processor for the ARPA Computer Network," Proceedings of the SJCC, May 1970, pp 551-567.

15. Anslow, N. G. and J. Hanscoff, "Implementation of International Data Exchange Networks," Computer Communications: Impacts and Implications, S. Winkler (ed.), Washington, October 1972, pp 181-184.

16. McKenzie, A., "HOST/HOST Protocol Design Considerations," (INWG Note 16, NIC 13879), January 1973.

17. Kahn, R. E., "Resource Sharing Computer Communication Networks," Proc. IEEE, Nov. 1972.